Mercury: Properties and Design of a Remote Debugging Solution using Reflection

Nick Papoulias, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse, Marcus Denker

JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Mercury: Properties and Design of a Remote Debugging Solution using Reflection

Nick Papoulias[a] Noury Bouraqadi[b] Luc Fabresse[b]
Stéphane Ducasse[a] Marcus Denker[a]

a. RMoD, Inria Lille Nord Europe, France
http://rmod.lille.inria.fr

b. Mines Telecom Institute, Mines Douai
http://car.mines-douai.fr/

Abstract Remote debugging facilities are a technical necessity for devices that lack appropriate input/output interfaces (display, keyboard, mouse) for programming (e.g., smartphones, mobile robots) or are simply unreachable for local development (e.g., cloud-servers). Yet remote debugging solutions can prove awkward to use due to re-deployments. Empirical studies show us that on average 10.5 minutes per coding hour (over five 40-hour work weeks per year) are spent for re-deploying applications (including re-deployments during debugging). Moreover current solutions lack facilities that would otherwise be available in a local setting because it is difficult to reproduce them remotely. Our work identifies three desirable properties that a remote debugging solution should exhibit, namely: *run-time evolution, semantic instrumentation and adaptable distribution*. Given these properties we propose and validate Mercury, a remote debugging model based on reflection. Mercury supports run-time evolution through a causally connected remote meta-level, semantic instrumentation through the reification of the underlying execution environment and adaptable distribution through a modular architecture of the debugging middleware.

Keywords Remote Debugging, Reflection, Mirrors, Run-Time Evolution, Semantic Instrumentation, Adaptable Distribution, Agile Development

1 Introduction

More and more of our computing devices cannot support an IDE (such as our smartphones or tablets) because they lack input/output interfaces (keyboard, mouse or screen) for development (*e.g.,* robots) or are simply locally unreachable (e.g., cloud-servers). Remote debugging is a technical necessity in these situations since targeted devices have different hardware or environment settings than development machines.

Nick Papoulias, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse, Marcus Denker. *Mercury: Properties and Design of a Remote Debugging Solution using Reflection.*
In Journal of Object Technology, vol. V, no. N, 2014, pages M:1–0. doi:10.5381/jot.201Y.VV.N.aN

Yet remote debuggers can prove awkward to use due to re-deployments in-between remote debugging sessions. Given that testing and debugging are estimated to cover roughly 50% of the developement time [Bei90b] and that validation activities - such as debugging and verification - cover 50 % to 75 % of the total developement costs [BH02], lenghty re-compilations and re-deployments of applications to remote targets can indeed have a negative impact on the development cycle. Empirical studies show us that on average 10.5 minutes per coding hour (over five 40-hour work weeks per year) are spent for re-deploying applications (including re-deployments during debugging) [Zer11]. This means that the specific facilities that a remote debugging solution offers (e.g.,. for incremental updating) during a remote debugging session can have a significant influence on productivity.

Moreover current solutions lack facilities that would otherwise be available in a local setting because its difficult to reproduce them remotely (e.g.,., object-centric debugging [RBN12]). This fact can impact the amount of experimentation during a remote debugging session - compared to a local setting. Empirical studies on software evolution [SMDV06] support this argument, identifying cases where the relationship or the interaction between two or more objects at runtime are more relevant to the programmer than the examination of execution at specific lines of code. Although emulators for target devices can help in this case, they are not always available, and are often of limited accuracy when sensory input or actuators are involved.

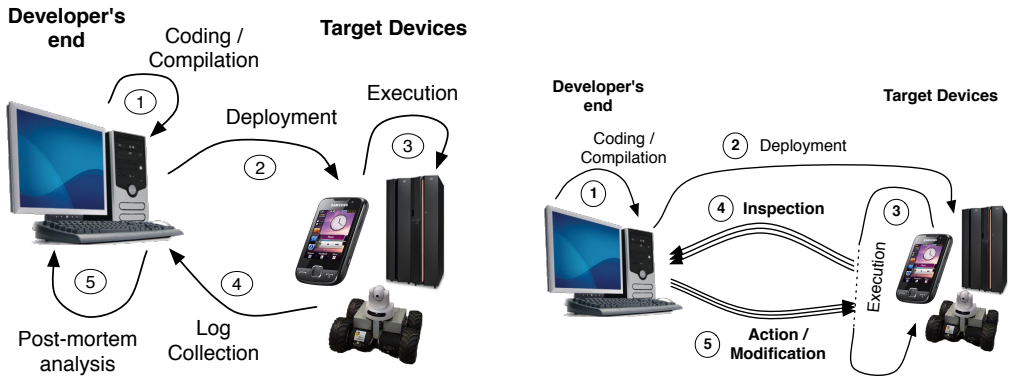
Finally remote debugging targets are becoming more and more diverse. Designing a debugging communication middleware to cover a wide range of applications (from mobile apps to server side deployments) that indeed operate under very different assumptions can be challenging. This fact has been pushing debugging middleware towards more and more extensible solutions.

In this work we identify three desirable properties for remote debugging: *run-time evolution, semantic instrumentation and adaptable distribution*. Given these properties we first evaluate existing solutions and then propose a live model for remote debugging that relies on reflection and more specifically on the concept of Mirrors [BU04]. We show how our model leverages both structural and computational reflection to meet the properties we have identified and present its implementation in the Pharo language [BDN+09] discussing the trade-offs that remote debugging imposes on reflective facilities. Finally we validate our proposal by exemplifying remote debugging techniques supported by Mercury's properties, such as *remote agile debugging* and *remote object instrumentation*.

The contributions of this paper are the following:

- The identification of three desirable properties for remote debugging solutions.
- The definition of a remote meta-level and infrastructure for remote debugging that can exhibit these properties.
- A prototype implementation of our model and its validation.

Our work is organized as follows: we begin by providing definitions and motivation for *remote debugging* in Section 2. We then introduce the properties for remote debugging solutions (Section 3) that we have identified. Then in Section 4 - given these properties - we evaluate existing solutions. Section 5 presents our proposed model: Mercury. Section 6 details our prototype implementation of the Mercury model. Section 7 presents the experimental setting and validation of our proposal. Finally Section 8 concludes our work and presents future perspectives.



(a) Debugging with *Post-Mortem* Analysis

(b) Remote debugging and interaction with a live execution

Figure 1 – Log-based Static Debugging vs. Remote Debugging

2 Background & Terminology

We provide working definitions for debugging that depend loosely on those given by Zeller [Zel05] and Sommerville [Som01] respectively:

Debugging *is a two phases process via which a programmer: i) Relates a failure in the observational domain of a program to a defect [Bei90a, Hum99] in the program's code and ii) Subsequently validates the elimination of a defect by applying a fix in the program's code relating it to a success in the observational domain.*

Debugger *A debugger is an additional process of the execution domain of a program used for run-time analysis. The debugger acts upon the process that is being debugged (*i.e., *the debuggee), making a subset of the execution domain of a program part of the observational domain.*

Remote Debugging *Is the process of debugging, in the context where the process of the debugger runs on a different machine than that of the debuggee.*

Figure 1(a) shows how developers can debug a target in absence of any support. Coding and compilation (step 1 in Figure 1(a)) need to be done on a developer machine, that is supposed to provide an IDE. Once the software is compiled, it is deployed (step 2) and executed (step 3) on the target. Next, the execution log is collected and transferred to the developer's machine (step 4). Last, the developer can perform a *post-mortem* analysis of the execution log (step 5) to find out hints about defect causes.

When a problem arises during execution, the developer only relies on the log verbosity to identify the causes during the *post-mortem* analysis (step 5). If the log is too verbose, the developer might be overwhelmed with the amount of data. Conversely, limited logging requires to go again through a whole cycle, after adapting the code just for collecting more data. This is due to the static nature of logs whose content is determined at the coding and compilation step (step 1). These cycles for re-compilation and re-deployment are time consuming and make debugging awkward. Nevertheless logging is the most widespread technique [Zel05] for "manual" debugging since in its simplest form (of *printf debugging*) is accessible even to inexperienced developers and can have little to zero infrastructure requirements.

In Figure 1(b) we show the different steps of a *remote debugging* process. Steps 1 (coding) and 2 (deployment) are the same as before. However, the execution (step 3) is now

interruptible. This *interruption* is either user-generated (the developer chooses to freeze the execution to inspect it) or is based on predetermined execution events such as exceptions. Steps 4 and 5 represent the remote debugging loop. This loop takes place at execution time and in the presence of the execution context of the problem which can be inspected and modified. Step 4 represents the remote inspection phase, where information about the current execution context is retrieved from the target. While in step 5 we depict the modification phase where the developer can provide further user-generated interruption points (breakpoints, watchpoints, etc.), alter execution and its state (step, proceed, change the values of variables), or incrementally update parts of the code deployed in step 2 (save-and-continue, hot-code-swapping). Several loops can occur during the execution depending on the developers' actions (step, proceed, user-generated interruptions) and on execution events (exceptions, errors, etc.). Having the ability to introspect and modify a live execution (without loosing the context) is a major advantage compared to analyzing static logs.

3 Desirable Properties of Remote Debugging Solutions

In this Section, we present three desirable properties for remote debugging solutions that we have identified, namely: *run-time evolution*, *semantic instrumentation* and *adaptable distribution*. We introduce and discuss each property individually based on a typical software stack for remote debugging.

As we depict in Figure 2 the target device (on the right) that runs the debugged application must provide a middleware for communication and run-time debugging support for examining processes, the execution stack, the system's organization, introspection of instance and local variables, etc.. On the other hand, the developer machine must provide a middleware layer, debugging tools, but also a model of the running application that describes the application running on the target (*e.g.,* source code or breakpoints).

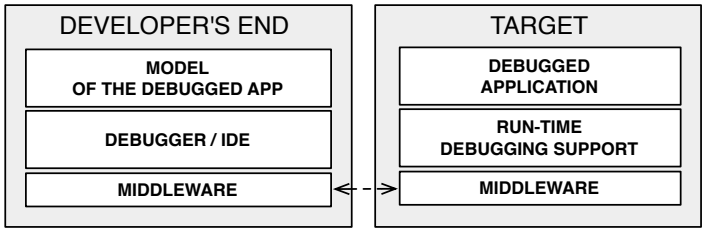


Figure 2 – Software Entities Involved in Remote Debugging.

3.1 Run-Time Evolution

Run-time evolution is the ability to *dynamically* inspect and change the target's application code and state. By dynamically here we mean that inspections and changes on the target can be performed while the application is running.

In Figure 2, there is an implicit relationship between the model of the debugged application (on the developer's end), and the state of the debugged application (on the target). This relationship can be either *static* or *dynamic*, depending on whether a change in either one of them updates the other. When a remote debugging solution supports run-time evolution, this relationship is dynamic.

The fact that the target application does not need to be restarted to be debugged and evolved allows developers to:

- Track the origins of bugs and fix them without losing the execution context.
- Fix flaws [Zel05] from within the debugger. Flaws are architectural bugs that are not associated with a specific location in the source file and require an architectural update (removing or adding new code) in order to be addressed.
- Increase productivity while debugging applications with a long startup time.
- Debug critical applications (*e.g.,* server side applications) that cannot be restarted.
- Experiment with heisenbugs [Gra86] as they are observed.

In OO languages developers should be able to evolve every organizational module of the target application while debugging. These changes ought to include:

Add/Rem Packages The ability to introduce new packages [1] and remove existing ones.

Add/Rem Classes/Prototypes The ability to introduce new classes or prototypes and to remove existing ones.

Hierarchy/Delegation Editing The ability to add/remove superclasses or edit a delegation chain.

Add/Rem Methods The ability to introduce new methods and to edit or remove existing ones.

Add/Rem Fields The ability to add/remove fields to a class or prototype.

3.2 Semantic Instrumentation

With the term *semantic instrumentation* we refer to the ability of a debugging solution to alter the semantics of a running process to assist debugging. Instrumentation is the underlying mechanism through which breakpoints and watchpoints are implemented. A debugging solution *instruments* the running process to halt at specific locations in the code, or when specific events occur (such as variable access) to either return control to the debugging environment or to perform predetermined checks and actions (such as breakpoint conditions).

In OO languages developers should be able to halt and inspect the running program both at specific locations in the source code and on specific semantical events that involve objects. In literature these events are referred to as *dynamic reification categories* [RC00]. These categories are a set of operations that can be thought of as *events* which are required for object execution [McA95][RRGN10].

Taking into account these semantic events, instrumentation categories for debugging should at least include:

Statement Execution The ability to halt at a specific statement or line in the source code.

Method Execution The ability to halt at a specific method in the source code upon entry. This is an event that can be triggered by multiple objects that share the same method (either through inheritance or through delegation).

Object Creation The ability to halt at object creation of specific classes or prototypes.

[1] We use the term *package* in its more general form here *i.e.,* as a named collection of related functionality

Field Read The ability to halt when a specific field of *any* instance of a class or prototype is read.

Field Write The ability to halt when a specific field of *any* instance of a class or prototype is written.

Object Read The ability to halt at *any* read attempt on a specific object.

Object Write The ability to halt at *any* write attempt on a specific object.

Object Send The ability to halt at any message send *from* a specific object. This is an event that is triggered when any method is invoked from withing the scope of a specific object.

Object Receive The ability to halt at any message send *to* a specific object.

Object as Argument The ability to halt whenever a specific object is passed as an argument.

Object Stored The ability to halt whenever a new reference to a specific object is stored.

Object Interaction The ability to halt whenever two specific objects interact in any way. This is a composite category as defined in [RRGN10]. It can be seen as a composition of object receive, send and argument categories.

3.3 Adaptable Distribution

As seen in Figure 2 remote debugging requires a communication middleware. Designing a communication solution for debugging to cover a wide range of applications that operate under very different execution environments can be challenging since debugging targets are becoming more and more diverse (ranging from small mobile devices to large server deployments on the cloud). Adaptable middleware solutions are well-studied in literature [KCBC02] and offer the ability of flexible configuration during runtime [RKC01] from which remote debugging frameworks can benefit. For example targets with different resources (memory, processing power, bandwidth) may require different serialization policies. While others such as server applications may require different security policies when they are being debugged through an open network.

We distinguish the following four categories of distribution support for debugging solutions, in ascending order of adaptability:

No-Distribution (-) The debugging solution does not support remote debugging.

Fixed-Middleware (+) The debugging solution supports remote debugging via a dedicated and fixed protocol which cannot be easily extended.

Extensible Middleware (++) The debugging solution supports remote debugging via a general solution for distributed computing (such as an object request broker) which can be extended, such as CORBA or DCOM.

Adaptable Middleware (+++) The debugging solution supports remote debugging via a general solution for distributed computing which can be extended and adapted at runtime [DL02].

4 Evaluation of Existing Solutions

We now study existing debugging solutions of major OO languages in current use today (Java and other OO JVM-based languages (JPDA) [Ora13b, Ora13a], C# (.NET Debugger) [Mic12b], C++ and Objective-C (through Gdb) [RS03]) and Javascript [Mic12a] as well as dynamic languages with *live programming* support (such as Smalltalk and its debugging model [LP90]) taking also into account bleeding-edge technological achievements [Zer12, WHV+12, PTP07] and very recent research results [WWS10, BNVC+11, RBN12].

4.1 Existing Solutions

JPDA/JVM Java's debugging framework stack is JPDA [Ora13b] and it consists of a mirror interface (JDI) [Ora13a, BU04], a communications protocol (JDWP) and the debugging support on the target as part of the virtual-machine's infrastructure (JVM TI). The application on the target machine must be specifically run with debugging support from the VM (the JVM TI) for any interaction between the client and the target to take place. JPDA does not provide facilities to dynamically update the target other then the hot-swapping of pre-existing methods. The communication stress is handled by the low-level debugging communication protocol (JDWP), whose specification is statically defined.

The debugging support for other JVM-based languages such as Scala [MO06] [Dra14], JRuby [Gra14] and Groovy [Eis14] rely on the debugging interfaces provided by JDI and provide all or a subset of the aforementioned facilities [Bru12].

JRebel and DCE The DCE VM [WWS10] and Jrebel [Zer12] are both modifications for the Java virtual machine that support redefinition of loaded classes at runtime. Although these modifications of the underlying VM are not a solution for debugging themselves, they do provide incremental updating facilities for remote targets. These modifications if used in conjunction with the JPDA framework can support the property of run-time evolution that we described in Section 3.

Maxine The Maxine Inspector offers debugging and dissasembling facilities for the experimental java virtual-machine Maxine [WHV+12]. Maxine is specialized for vm-level debugging (*i.e.,* for system programming) by virtue of targeting a meta-circular vm (*i.e.,* a vm written mostly in java itself). It provides a dual low-level / high-level view while debugging. This enables Maxine to support field watchpoints on objects regardless of whether the object has been moved or not by the gc and halt on semantical events related to garbage-collection. Maxine has experimental support for the JDWP communication protocol of JPDA but in contrast with JPDA suffers from performance issues [OL14] which are nevertheless a well-studied reality of meta-circular virtual-machines in general [USA05].

TOD TOD [PTP07] is an omniscient (*i.e.,* back-in-time) debugger targeting the Java language. TOD is usually used with an uninterrupted execution (without any breakpoints) in order to trace execution events. Although omniscient debugging is out of the scope of our work, TOD in particular relates to our discussion by providing an *event-driven* architecture for organizing and storing execution traces. This architecture utilizes object-oriented instrumentation events as the ones we describe in Section 3.2 including field write and message interception. TOD supports instrumentation through JVMTI and ASM [BLC02].

.NET As with Java, the main remote debugging solution for .NET provided through visual studio [Mic12b] pre-purposes a dedicated debugging deployment. In the developer's end the

model of the running application is again static, with the developer being responsible for providing the right sources and configuration files. In the case of .NET though the debugger can attach to a running remote process without loosing the context, provided that the static model for the application is available. Although the model in the developer's end is static, a limited form of updating is provided in the form of *edit-and-continue* [Mic12c] of pre-existing methods. There is currently no support for incremental updating of the target application with new packages, classes or methods although there are plans to add related features in the next major release of Visual Studio [2].

GDB For Obj-C remote debugging is provided through the gnu-debugger [RS03]. Gdb uses a dedicated process on the target machine called the *gdb-server* to attach to running processes. For full debugging support though the deployed application has either to be specifically compiled and deployed with debugging meta-information embedded on the executable or the static model for the running application has to be provided from a separate file. Gdb supports a limited form of updating through an *edit-and-continue* process of pre-existing methods by patching the executable on memory [RS03].

Smalltalk The most prominent example of an interactive debugger is the Smalltalk debugger [BDN+09, LP90, Gol84]. In Smalltalk the execution context after a failure is never lost since through reflection the debugger can readily be spawned as a separate process and access the environments' reifications for: *processes, exceptions, contexts* etc. Moreover it supports incremental updating in such a way that introducing new behavior through the debugger is not only possible but is actually advised [BDN+09]. Indeed incremental updating through debugging encourages and supports agile development processes, and more specifically Test Driven Development (TDD) [ABF05]. In addition both the debugging and the reflecting facilities of Smalltalk are extensible. On the one hand the debugger model is written itself in Smalltalk. On the other hand the Smalltalk MOP is readily editable from within the system itself. Illustrative examples of MOP extensions in Smalltalk are given from Rivard in [Riv96]. In terms of distribution, middleware proposals like Dist. Smalltalk [Ben88] and OpenCorba [Led99] can be in principle used to allow simple exception forwarding.

Bifrost In Smalltalk supporting advanced debugging techniques through instrumentation is illustrated in the Bifrost reflection framework [RRGN10] and through object-centric debugging [RBN12]. Bifrost is an extension to the Smalltalk MOP that relies on explicit meta-objects to provide sub-method [DDLM07] and partial behavioral reflection [TNCC03]. Bifrost is implemented through dynamic re-compilation of methods. Method invocations are intercepted using *reflective methods* [Mar06] similarly to the *wrappers* abstraction introduced by Brant et al. [BFJR98]. These are subsequently recompiled using AST meta-objects that control the generated bytecode. With Bifrost intercession techniques such as the explicit interception of variable access, is made available at the instance level.

JS/Rivet In–browser debugging of client-side web-applications is out of scope of our current work. Nevertheless we surveyed the debugging tools of the following web-browsers: IE [Mic14], Mozilla [Moz14] and Chrome [Goo14] in terms of the features discussed in Section 3. These solutions apart from classic debugging facilities for execution control and breakpoints, offer *event-driven* debugging facilities for the DOM (Document Object Model [WAC+98]) and some limited form of live-editing for slots and pre-existing methods.

Our own work is more related to the debugging facilities of standalone javascript applications (like those of Node.js [Joy14]) which have nevertheless very rudimentary support

[2] https://www.visualstudio.com/en-us/downloads/visual-studio-2015-ctp-vs

for execution control and breakpoints. The experimental solution of Rivet from Microsoft [Mic12a] comes closer than other solutions for JS to what we have described in Section 3. It supports run-time evolution through live-patching and has an easily extensible communication architecture build on top of HTTP.

AmbientTalk AmbientTalk is an ambient-oriented programming language dedicated to mobile ad-hoc networks. AmbientTalk's reflective model supports mirages [MCTT07] which are an extension to the mirror model adding implicit reflection capabilities. Through implicit reflection AmbientTalk extends the contemporary API of mirrors with the following three methods: a *doesNotUnderstand(selector)* protocol for message sends and two implicit hooks for object marshalling (pass() and resolve()) [MVCT+09]. These hooks can be used during a debugging session to support some of the object-oriented execution events we describe in Section 3, such as instantiation events, slot editing and interception [Cut14]. REME-D (the debugging framework of AmbientTalk) supports classical features of remote debuggers, such as step-by-step execution, state introspection and breakpoints and builds upon the reflective capabilities of AmbientTalk to support asynchronous message breakpoints and epidemic debugging [BNVC+11]. These features although domain specific to mobile ad-hoc networks are nevertheless supported by a communication paradigm (AmOP) that is very robust and adaptable.

4.2 Comparison

In this Section we compare existing solutions in terms of *run-time evolution,semantic instrumentation and adaptable distribution*. Table 3 of Appendix A presents our findings in details (per sub-properties) while Table 1 (also part of Section 5) gives a summary per-solution.

4.2.1 Run-Time Evolution

As seen in Table 3 debugging environments of mainstream OO languages (JPDA, .Net Debugger, Gdb) do not support run-time evolution with the exception of a *save-and-continue* facility for pre-existing methods. In the case of Gdb method hotswapping can lead to inconsistencies [Zel05] since it is supported through memory patching, which is a blind process that replaces execution instructions in memory, without knowledge of the underlying semantics of the language. In the Java world recent developments (through Jrebel and DCE) provide full support for this property as do more dynamic environments such as Smalltalk, Bifrost and Rivet.

4.2.2 Semantic Instrumentation

In Table 3 we also do a comparison in terms of semantic instrumentation and its sub-properties as they were defined in Section 3. We have also included a last category marked as *condition/action* that describes whether in all instrumentation events the debugging solution can support **both** user-generated checks **and** code in order to provide a more fine-grain control. As an example we can consider a conditional breakpoint that is able to execute user specified actions when triggered.

As we can see from our comparison, Bifrost is the front-runner of instrumentation with all mainstream solutions supporting only plain breakpoints and watchpoints while other – more experimental solutions – having a partial coverage. Bifrost though lacks an *Object Stored* event which is useful for following an object's reference propagation and counting. Finally only Bifrost and Gdb provide support for both conditions and actions on instrumentation events.

4.2.3 Adaptable Distribution

Finally Table 3 presents our comparison in terms of distribution. Solutions are marked with - for not supporting distribution, + for supporting distribution through a fixed-middleware, ++ for an extensible middleware and +++ for an adaptable middleware.

As we can see no general purpose solution supports an adaptable middleware, besides AmbientTalk which is an environment tailored to mobile ad-hoc networks. Nevertheless even in mainstream solutions we notice a trend towards more and more dynamicity. The .NET debugging framework for example follows in the comparison using a general purpose and extensible communication solution (DCOM) [Mic13] together with Rivet's HTTP-based protocol. We should note here that in the case of Smalltalk (which does not have a dedicated solution for remote debugging), there were some efforts in the past to support remote development (including simple exception forwarding for debugging) in Cincom Smalltalk based on [Ben88], which were discontinued.

4.2.4 Comparison Overview

In Table 1 we present an overview of our comparison in terms of all properties that were described in Section 3:

Property	JPDA/JVM	.NET	GNU-DEBUGGER	DCE	JREBEL	SMALLTALK
Run-Time Evolution	**+ (1/6)**	**+ (1/6)**	**+ (1/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**
Sem. Instrumentation	**+ (4/13)**	**+ (4/13)**	**+ (5/13)**	**+ (4/13)**	**+ (4/13)**	**+ (3/13)**
Ad. Distribution	**+ (fixed)**	**++ (extensible)**	**+ (fixed)**	**+ (fixed)**	**+ (fixed)**	**- (no)**
Property	JS/RIVET	TOD	AMBIENTTALK	MAXINE	BIFROST	MERCURY
Run-Time Evolution	**+++ (6/6)**	**+ (1/6)**	**+ (2/6)**	**+ (1/6)**	**+++ (6/6)**	**+++ (6/6)**
Sem. Instrumentation	**+ (2/13)**	**++ (7/13)**	**++ (8/13)**	**++ (7/13)**	**+++ (12/13)**	**+++ (13/13)**
Ad. Distribution	**++ (extentible)**	**+ (fixed)**	**+++ (adaptable)**	**+ (fixed)**	**- (no)**	**+++ (adaptable)**

Table 1 – Comparison Overview for Existing Solutions and Mercury (Section 5)

As we can see from Table 1 debugging solutions based on reflection (such as Smalltalk and Bifrost in the local scenario) offer the most complete solutions in terms of run-time evolution and semantic instrumentation, but lack support for adaptable distribution (as in the case of AmbientTalk). On the other hand solutions of mainstream OO languages (JPDA, .Net Debugger, Gdb) and their extensions (Jrebel, DCE) lack support for either run-time evolution or instrumentation (or in some cases both). Other solutions (Rivet, TOD, AmbientTalk, Maxine) have partial support for some but not all of our properties. There is no solution that meets all our criteria in a satisfactory way.

5 Our Solution: Mercury

Our solution proposes a model of the debugged application (cf. Figure 3 (1)) that is dynamic and acts as a meta-level for target applications. It relies on specific meta-objects known as *Mirrors* defined by Bracha and Unghar as "*intermediary objects [...] that directly correspond to language structures and make reflective code independent of a particular implementation*" [BU04]. Mirrors are located on the developer's side. They are causally connected to the debugged application and support *run-time evolution*. The run-time debugging support on the target (cf. Figure 3 (2)) reifies the underlying execution environment to support *semantic instrumentation*. Finally our middleware follows a modular architecture to be adaptable even during runtime (cf. Figure 3 part (3)).

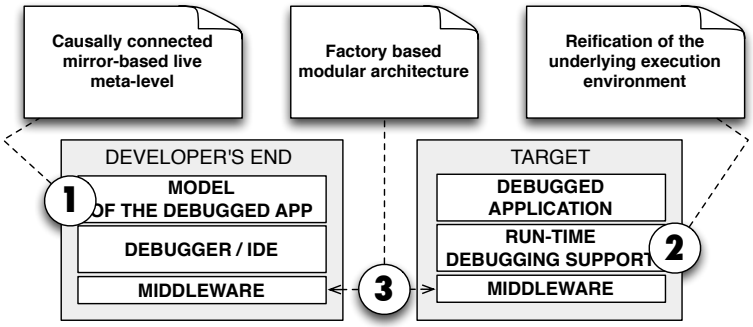


Figure 3 – Overview of our Solution

The rest of this section describes our solution based on this *live meta-level* and how it supports the three properties which were identified in Section 3.

5.1 The Core Meta-Level

The meta-level located on the development machine (left part of Figure 4) is a set of mirrors that reflect on objects (e.g., instance of the class Point) on the target side (right part of Figure 4). The target machine also includes support for reflection and debugging. This is the role of the package RTSupport that includes the RunTimeDebuggingSupport class (our remote facade).

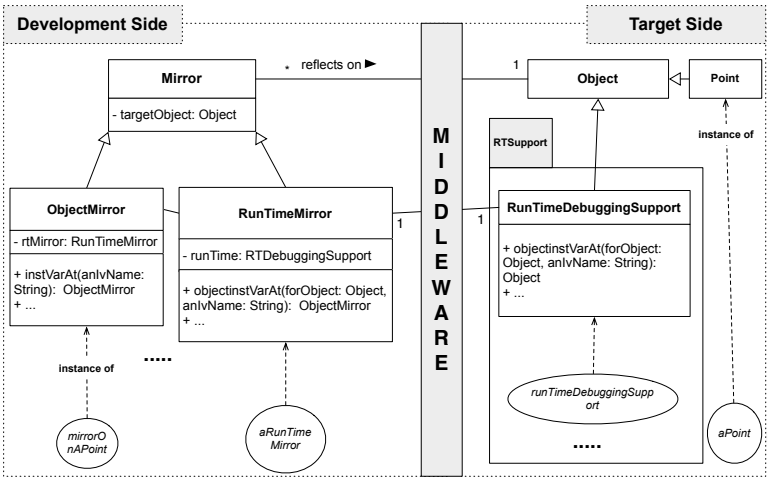


Figure 4 – Our core model

On the left side of Figure 4, we depict the 3 core classes of our meta-level. The root is the Mirror class, that declares the targetObject field. So, every mirror holds a remote reference to one object on the target. Nevertheless, an object on the target can be reflected by multiple mirrors on the development side.

Both on the developer's end and on the target, a unique object is responsible to handle all communications to the other side. This object is an instance of RunTimeMirror on the developer's end and an instance of RunTimeDebuggingSupport on the target. On the developer's end, all mirrors can retrieve this object in their inherited field named rtMirror. The API of

the RunTimeMirror is completely equivalent to the one of RunTimeDebuggingSupport with one crucial difference: *each call to the target side results in a mirror or a collection of mirrors being returned to the developer's side* ensuring the encapsulation of the remote debugging facilities. This *cascading encapsulation* between calls to a mirror object is equivalent to the transitive wrapping mechanism described by C. Teruel et al. [TCD13] for proxies, only in this case an entire remote environment is being wrapped rather than a local object-graph.

To show how communication and reflection is handled between the development machine and the target, consider the example of the mirror mirrorOnAPoint and its target object aPoint on Figure 5. Suppose that the developer wants to get the class of aPoint. To perform this operation, the IDE sends the getClass message to mirrorOnAPoint. As a result, mirrorOnAPoint sends the getClass(targetRef) message to aRunTimeMirror passing as a parameter the remote reference that it holds. Then, aRunTimeMirror invokes through the middleware the corresponding getClass(targetRef) method on runTimeDebuggingSuppport located on the target. The runTimeDebuggingSuppport retrieves the class Point and answers it back through the middleware. The class on the target is retrieved either via local reflection or through direct vm-support provided by the RunTimeDebuggingSupport class of Figure 4. On the developer side, aRunTimeMirror receives a remote reference on the Point class, and creates a new mirror on the remote class. It is this mirror on the Point class that is returned back to mirrorOnAPoint.

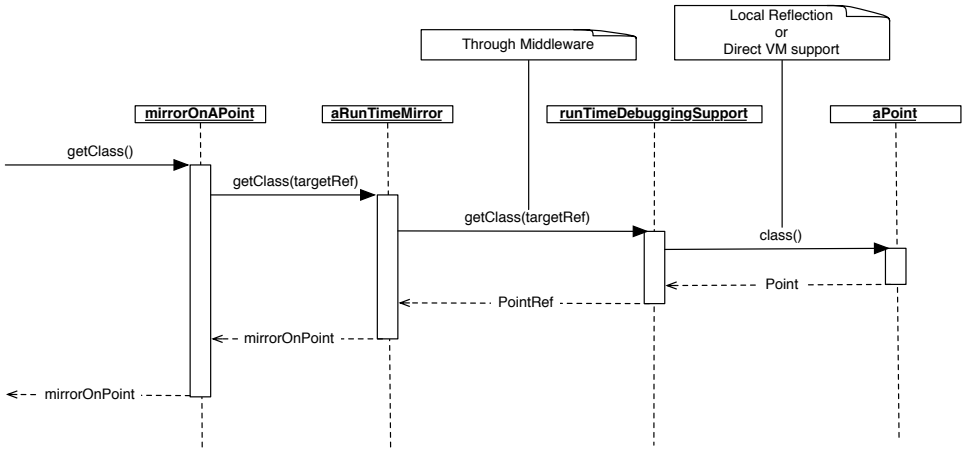


Figure 5 – Sequence Diagram detailing Remote Reflection with Mercury

Communication can also be initiated by the target as shown in Figure 6 to trigger updates of the remote meta-level on the developer's side. For example when a new exception is thrown on the target (right side of Figure 6) the asynchronous newException(exceptionRef) message is send carrying a remote reference to this newly raised exception. As before aRunTimeMirror will receive the remote reference and will create a new exception mirror. It is this exception mirror that will be forwarded to interested listeners who have registered through the registerListener(aListener) message of the run-time mirror. Typically a listener will be an instance of the EnviromentMirror class (see Figure 7) through which interrupted processes and unhandled exceptions are accessed.

5.2 Run-Time Evolution

To support run-time evolution, the model of the debugged application on the developer's end and the state of the debugged application on the target (cf. Figure 2) needs to be *causally connected*. This means that an arbitrary change in either one of them should update the other.

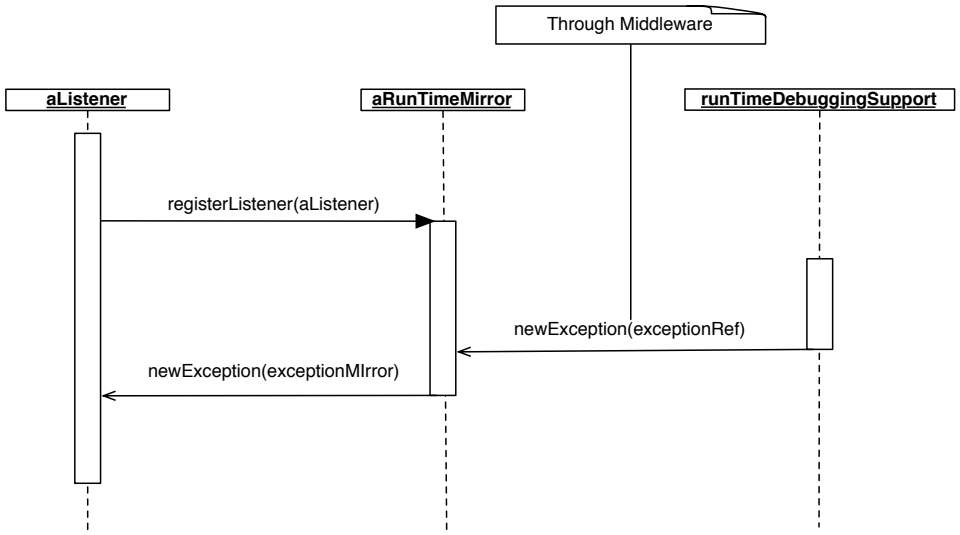


Figure 6 – Asynchronous communication initiated by the target

We describe how our model supports this property through the class hierarchy and the API of our meta-level (starting from ObjectMirror). Figure 7 depicts 8 core classes of our meta-level which are divided into two groups: the ones that reify the structure of the debugged application (*structural reflection*) and the ones that reify the computation (*computational reflection*) [Fer89, Mae87].

In our model, both *structural reflection* and *computational reflection* are *causally connected* to the other side. For *structural reflection*, this means that the addition of a new package, a new class or method through mirrors, etc. in the development side results in a structural update of the running application on the other side. These 8 core classes depicted in Figure 7 define an API that supports run-time evolution. Instances of these classes reflect on remote objects on the target and all of their methods can be executed while the application is running.

ObjectMirror. An ObjectMirror enables retrieving information from the object reflected such as its class, reading/setting its fields or sending new messages to it but also changing its class (setClass).

EnvironmentMirror. It is the entry point mirror to the target application depicting the remote environment as a whole. Through the environment mirror globals are read/written, loaded packages are retrieved, interrupted processes and unhandled exceptions are accessed, code is evaluated (evaluate) and packages can be created, removed, or edited (newPackage, removePackage, etc.).

PackageMirror. A package mirror reflects on loaded packages on the target application. This mirror gives access to package's meta-information such as its name and the classes it contains. Classes can also be added or removed using the methods newClassNamed and removeClassNamed.

ClassMirror. Through a class mirror the name, superclass, fields, methods and enclosing package of the reflected class can be retrieved. The superclass can be changed, new instance variables and methods can be added/removed or edited (setSuperClass, addInstVarName, deleteInstVarName, addMethod, deleteMethod, etc.).

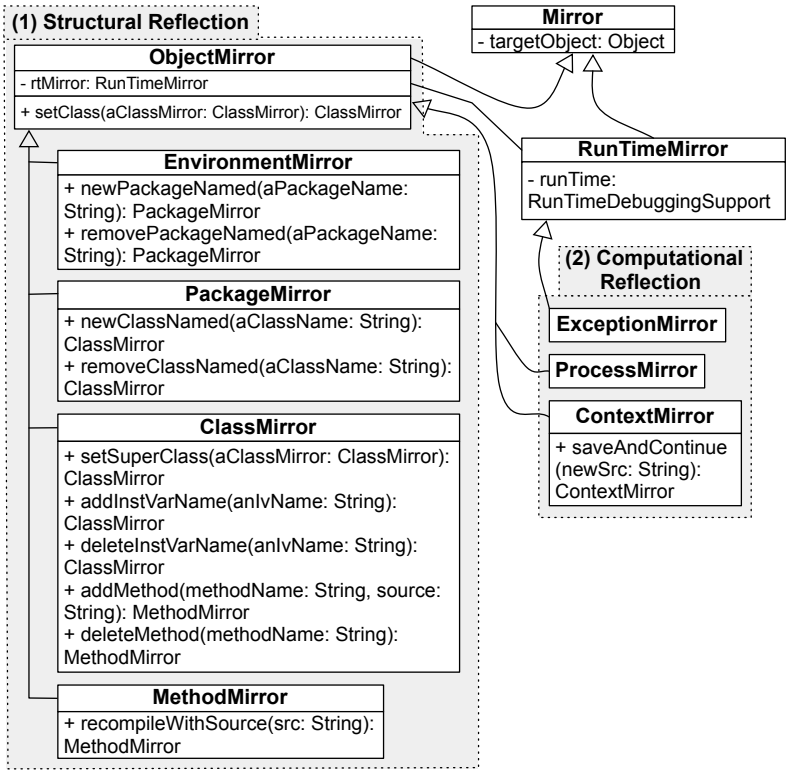


Figure 7 – Core classes and API for supporting Run-Time Evolution

MethodMirror. Apart from retrieving the name, source or class membership of a Method, the developer can edit a method in place (recompileWithSource).

ProcessMirror. It allows one to retrieve meta-information on a process such as its stack and manipulate the execution flow.

ExceptionMirror. It is the reification of exceptions on the target. Through an exception mirror the description of an unhandled exception can be retrieved, as well as the process that it occurred and the offending execution context.

ContextMirror. It is the reification of a stack frame (context) on the target application. Through a contextMirror its process, method, receiver and sender can be retrieved, temporaries and arguments of the invocation can be read/written, its execution can be restarted but also the method that was invoked and created the context can be edited before continuing the execution (saveAndContinue).

5.3 Semantic Instrumentation

Semantic Instrumentation in our model is supported through *intercession*. Specifically the underlying execution environment is reified inside the run-time environment of the target as to be able to control the semantics of a running process.

The model of our solution uses the following patterns:

The observer [ABW98] An observer defines a dependency between an object and its *dependents*, so that the dependents are notified for state changes on that object.

The implicit meta-object [Mae87] Implicit meta-objects are meta-objects that are invoked automatically by the underlying execution mechanism.

Objects can be instrumented either to perform user-generated conditions and actions upon invocation of specific events (e.g., RunTimeDebuggingSupport»objectOnReceive) or to halt the process on those specific events (e.g., RunTimeDebuggingSupport»objectHaltOnReceive).

Figure 8 depicts the reification of the Interpreter (the underlying execution environment) which acts as our observer, connecting instances of Object (regular objects) to instances of ImplicitMetaObject (dependents). Whenever an event of interest is being applied to an object (such as a message send) the underlying execution mechanism invokes the Interpreter reification, which in turn notifies the ImplicitMetaObjects. The Interpreter resolves the relationship between objects and meta-objects through the MetaEnvironment, which acts as an environment dictionary for the meta-level. The MetaEnvironment provides a one-to-one mapping between objects and meta-objects.

Implicit meta-objects when notified, will invoke a callback (class Closure in Figure 8) which can be either a local callback or a remote callback from the developer's end. The RunTimeDebuggingSupport maintains a reference to the Interpreter reification to register these callbacks coming from mirrors on the developer's side. Thus our implicit meta-objects extend our mirror model in order to add implicit reflection capabilities as in Mostinckx et al. [MCTT07] (see also *AmbientTalk* on Section 4). In our case though these meta-objects implement all additional implicit events for remote debugging described in Section 3 which are made possible by our reification of the underlying execution environment.

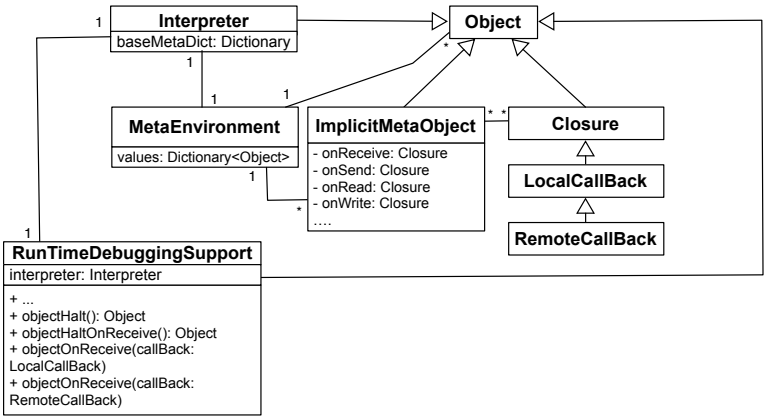


Figure 8 – Core classes for Instrumentation support in the Target

5.4 Adaptable Distribution

To support distribution via an adaptable middleware, we modeled our solution using the concept of the abstract Factory [ABW98], through which families of related objects can be assembled and parametrized at runtime. Our model for adaptable distribution is part of our proposed model for remote debugging. It is specifically tailored for this context and should not be considered a contribution on its own.

Figure 9 depicts the core classes of our model for distribution:

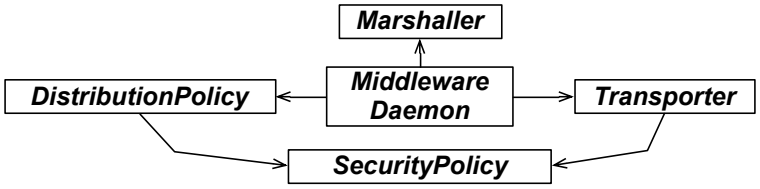


Figure 9 – Core classes of our adaptable middleware

Middleware Daemon This abstract class defines methods for the orchestration (assembling) and initialization of our middleware, acting as an Abstract Factory. It is also responsible for loading the RTSupport package (cf Figure 4) on the target upon the successful authentication of a client.

Transporter The concrete subclasses of this abstract class handle the actual communication between peers. Different transporters can support different communication protocols (e.g., tcp, udp or web-sockets)

Marshaller The marshaller (through its concrete subclasses) is responsible for serializing and materializing information, passed through the connection. Different marshallers can support different transcoding algorithms to fit the needs of the debugging context (e.g., serializing to xml, json or binary-form).

Distribution Policy This class (through its concrete subclasses) decides how specific objects or group of objects will be distributed among peers. Options can include: full serialization, shallow serialization, proxying, etc..

Security Policy The concrete subclasses of this abstract class are responsible for authentication and for restricting access (either message sending or distribution) for specific instances or whole classes of objects.

5.5 Comparison with Existing Solutions

In this Section we compare the state-of-the-art debugging solutions (which we discussed in Section 4.2.4) with our work in terms of *run-time evolution, semantic instrumentation and adaptable distribution*.

As we can see from Tables 1 and 3 (Appendix A) our solution manages to cover all three properties that were identified in Section 3 being comparable only to the Bifrost framework (in the local scenario) in terms of run-time evolution and semantic instrumentation. In our case though these properties are brought to remote debugging through an adaptable middleware. In terms of distribution Mercury is only comparable to AmbientTalk and to a lesser extend to Rivet and the .NET debugging framework which both use an extensible (but not adaptable) communication middleware (DCOM) [Mic13].

Finally since both our solution and Bifrost are based on Smalltalk, we were also able to perform a micro-benchmark to compare the two, in terms of the overhead introduced by instrumentation. The benchmark is based on Tanter [TNCC03] and the Bifrost metrics are those reported in [Res12]. The benchmark measures the slowdown introduced by each solution for one million messages send to a test object when a) no instrumentation is present b) instrumentation is loaded but is disabled for this specific object and c) instrumentation is enabled on the test object of the micro-benchmark.

	BIFROST	MERCURY
No instrumentation	1x	1x
Disabled instrumentation	1x	1x
Enabled instrumentation	35x	8x

Table 2 – Instrumentation benchmark for Bifrost and Mercury

As we see in Table 2 for both solutions there is no overhead introduced when a specific object is not being instrumented, regardless of whether the solution is loaded into the environment. This is important for practical reasons so as to avoid slowing down the whole system while debugging. While instrumenting a specific object our solution introduces a significantly smaller overhead than Bifrost. We believe that this is due to the fact that our solution is based on the underlying virtual-machine rather than on byte-code manipulation as in the case of Bifrost.

5.6 Limitations

In contrast with the debugging solutions presented in Section 4 our solution has the following limitiations:

Distributed Applications Mercury is a model for *remote debugging* and can thus seperately debug the connected parties of a distributed application, as is usually the case with debugging remote server or client applications. Special debugging facilities for distributed applications - such as asynchronous breakpoints supported by REME-D [BNVC+11] - were outside the focus of our work.

VM-Debugging Also our solution concerns itself exclusively with *language-side* debugging, in contrast for *e.g.,* with Maxine [WHV+12]. Maxine targets a meta-circular language-vm system and can thus be used to debug both the Maxine virtual-machine as well as the java program running on top of it.

In-browser Debugging Finally Mercury cannot be used to debug in-browser client-side applications. We nevertheless surveyed several javascript debugging solution in terms of the properties discussed in Section 3.

6 Mercury's Implementation

6.1 Implementation Overview

Given our state-of-the-art survey in Section 4 we chose to implement Mercury in a platform that was as close as possible to our goals. We chose to implement a prototype[3] of our model (described in Section 5) in Pharo [BDN+09] and Slang [IKM+97]. Pharo is a reflective, object-oriented and dynamically typed programming environment that is inspired by Smalltalk. Slang is a subset of the Smalltalk syntax with procedural semantics that can be easily translated to C. In Figure 10 we show the different constituents of our implementation.

[3] http://ss3.gemstone.com/ss/Mercury-Prototype.html

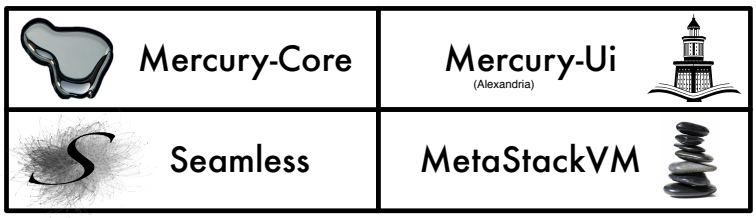


Figure 10 – Core parts of Mercury's Prototype

MetaStackVM Is a dedicated virtual-machine for debugging targets, that extends the reflective facilities of the standard Stack VM of Pharo [Mir08] in order to support intercession [Pap13].

Seamless Is our adaptable middleware that provides flexible communication facilities betweens peers during debugging sessions.

Mercury-Core Is the sub-project of Mercury that hosts the debugging meta-level and the debugging run-time support.

Mercury-UI Is a debugging front-end that exemplifies key functionalities of our solution.

All four part of our prototype implementation for Mercury are released under the MIT license [4].

6.2 Discussion: Implementation trade-offs

6.2.1 Supporting Run-Time Evolution

Implementors of our model have essentially two options for supporting run-time evolution through the RunTimeDebuggingSupport (depicted in the left side of Figure 4):

(a) Local reflection Local reflection on the target can be used to provide the corresponding API for run-time evolution. This solution is applicable to languages that already provide a rich set of local reflective facilities. It is also a portable and extensible solution since the debugging support is written in the same language as the target application.

(b) Virtual Machine support Debugging support on the target can be also hard-coded inside the virtual-machine of the target. This solution fits better with languages that do not support advanced reflective facilities on their own. It is also an attractive option for system debugging, in cases where core language reflection itself has to be debugged. This solution is less portable and extensible if it is not supported by the vendor of the target language.

In our prototype we used a combination of the two approaches mentioned above. Remote reflection on the instance level is separated from local reflection on the target and can thus support some limited form of system debugging. However, we also make use of local reflective facilities on the target for system-organization reflection (packaging meta-objects) and computational reflection (reifications of contexts and processes). Our implementation currently depends on the compiler on the target. Ideally, the developers' end compiler should be used and the target should not host a compiler itself to further minimize the footprint.

[4] http://opensource.org/licenses/MIT

6.2.2 Supporting Instrumentation

To support semantic instrumentation the following options can apply:

(a) Bytecode Manipulation The compiler can be used to re-compile part of the system to transparently introduce crosscuts that perform instrumentation checks (for message sending, field access, etc.). This solution has the disadvantage of instrumenting only static entities (such as classes or methods) and may perform poorly when specific objects (runtime entities) need to be instrumented. For example when instrumenting message sending on a specific object, all the methods of its class and its superclasses have to be re-compiled to introduce the crosscuts. On the other hand in the case of a self-hosted compiler this option favors portability.

(b) Virtual Machine support Instrumentation support on the target can be also hard-coded inside the virtual-machine of the target. This solution fits better with instrumentation of run-time entities, since the checking can be performed on the object itself while it is being interpreted by the underlying execution environment. Portability may be an issue in this case if instrumentation is not supported by the vendor.

In our prototype we supported instrumentation by extending the stack-based virtual machine of Pharo. We chose to provide virtual-machine support since our focus was on instrumenting run-time rather than static entities. Furthermore we did not wish to have further dependencies on the compiler of the target.

6.3 Discussion: Implementing Mercury in Java

We take the Java language as an example to investigate the feasibility of implementing Mercury in other languages. We discuss the technological prerequisites for implementors for each part of our model as was discussed in Section 5.

A causally connected dynamic meta-level for debugging that can support run-time evolution can be build for Java by combining the currently available debugging infrastructure found in JPDA [Ora13b] with the incremental updating facilities of the DCE VM project [WWS10] or those found in the JRebel vm-plugin [Zer12] (see also Section 4). Support for semantic instrumentation can be build on top of solutions for bytecode manipulation or reflective intercession for Java like those in Iguana/J [RC00], Reflex [TBN01], ASM [BLC02] or JavaAssist [CN03]. We should note here however that since Java is more static in nature - compared to Pharo - these frameworks should be able to inter-operate with the incremental updating support we discussed previously (DCE, JRebel) to apply the required adaptations at run-time as we do with Mercury. Supporting remote debugging through an adaptable middleware can be achieved in Java by substituting the static low-level debugging communication protocol (JDWP) of JPDA with a more dynamic and flexible middleware solution like Cajo [Cat14]. Another approach would be to reconcile core reflection with remote method invocation bypassing the restrictions imposed by the Java reflection API. This was proposed in the work of Richmond and Noble [RN01] through a set of carefully designed local and remote proxies.

7 Mercury's Validation

In this Section we first show some basic examples of Mercury in terms of API (sub. 7.1) and then continue by validating Mercury's properties in an experimental setting (sub. 7.2). Two case studies are considered involving remote debugging of multiple remote devices. The first

case study details how the property of interactiveness can be used to support a remote agile debugging paradigm (sub. 7.3). While the second shows how Mercury brings the idea of object-centric debugging in a distributed setting through remote object instrumentation (sub. 7.4).

7.1 Basic API Examples

7.1.1 Inspecting remote environments and accessing objects

On line 1 of Script 1, the current process on the developer machine uses the mirror factory Reflection to access an environment on the remote target at address minesdouai.fr:8080. On line 2 a package meta-object is accessed named: #Graphics-Primitives, then on line 3 a class meta-object inside this package named: #Point is retrieved. On line 4 a new instance of the class #Point is created on the target and its corresponding meta-object is returned on the developer's machine. Finally on line 5 the instance variable named x of this newly created object is set to a new value.

Example Script 1: **Inspecting a remote environment and editing remote objects**

```
1 anEnvironmentMirror:= Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 aPackageMirror := anEnvironmentMirror packageNamed: #'Graphics-Primitives'.
3 aClassMirror := aPackageMirror classNamed: #Point.
4 anObjectMirror := aClassMirror newInstance.
5 anObjectMirror instVarAt: #x put: 100.
```

7.1.2 Handling remote exceptions

In Script 2, as before the remote environment is accessed through our mirror factory (line 1). Then on line 2 an expression is evaluated on the remote target: '3 / 0'. Then we show that if a remote exception occurs during the evaluation of an expression, a corresponding exception meta-object is returned and can be used by clients of our meta-level. In this case a LocalDebuggingClient class which makes use of our meta-level, is invoked with our remote exception meta-object.

Example Script 2: **Handling a remote exception**

```
1 anEnvironmentMirror := Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 anEnvironmentMirror
3       evaluate: '3 / 0'
4       onRemoteExceptionDo: [:aRemoteExceptionMirror |
5             LocalDebuggingClient debug: aRemoteExceptionMirror]
```

7.1.3 Changing variables and controlling execution flow

On line 2 of Script 3, we access all interrupted processes (threads) on the remote target. These are all execution threads that have raised unhandled exceptions or have been interrupted for inspection by the developer. Then on line 3 we retrieve the top context on the stack of the first interrupted process. On line 4 we modify a temporary value inside that context. Then finally on line 5 we make the process to proceed with the interrupted execution.

Example Script 3: **Controlling execution flow**

```
1 anEnvironmentMirror := Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 interruptedProcessesMirrors := anEnvironmentMirror interruptedProcesses.
3 aContextMirror := interruptedProcessMirrors first topContext
4 aTempObjMirror := aContextMirror tempNamed: 'x' put: '3'.
5 interruptedProcessesMirrors first proceed.
```

7.1.4 Incrementally changing the target's code and state

On Script 4, we show how a developer can introduce new behavior in the targeted application (interactiveness). On line 2 we introduce an empty new package through our environment meta-object. From line 3 to 5, we add a new class in this package with two instance variables and one method named hypothesis. We instantiate this new class (line 6) and send a message to this new instance (line 7) through its meta-object. As before (on Script 3) if our message-send raises an exception, a corresponding exception meta-object will be returned which can be used by clients of our meta-level.

Example Script 4: **Incrementally updating the remote target to test a bug**

```
1 anEnvironmentMirror := Reflection on: RemoteEnvironment @ 'mines-douai.fr:8080'.
2 aPackageMirror := aRemoteEnvironment newPackageNamed:  #NewPackage.
3 aClassMirror := aPackageMirror newClassNamed:  #NewClass.
4 aClassMirror := aClassMirror ivs: { #x .  #y}.
5 aClassMirror := aClassMirror addMethod: 'hypothesis: aNumber ...'
6 anObjectMirror := aClassMirror newInstance.
7 anObjectMirror perform:  #hypothesis
8        withArguments: { 3 }
9        onRemoteExceptionDo: [:aRemoteExceptionMirror |
10              LocalDebuggingClient debug: aRemoteExceptionMirror]
```

7.1.5 Introducing breakpoints on execution events

In Script 5, we show how instrumentation can be used to alter semantics on the target application and provide facilities such as object-centric watchpoints. Especially in this example we show how halting on object creation can be achieved by conditioning the message receive event on a class.

On line 1 a class mirror is retrieved. On line 2 a remote callback is registered for instrumenting message sending on the remote class. This callback accepts one or more arguments that provide meta-information about the event, such as the name of the method being invoked. On line 3 a condition and an action are set within the callback. Specifically when the message *new* (responsible for object creation) is sent to the remote class, the class that triggered the event (i.e *reifications trigger*) will cause the remote process to halt, effectively producing a watch-point on object creation.

Example Script 5: **Instrumentation of object creation**

```
1 aClassMirror := aPackageMirror classNamed: #Point.
2 aClassMirror onReceive: [:reifications |
3       reifications message selector = #new ifTrue: [reifications trigger halt] ]
```

7.1.6 Distribution

In Script 6, we show how we can adapt the middleware's serialization policy at runtime while debugging. On line 1 we retrieve a class mirror (on the class #Class) and then on line 2, we ask the class mirror for an object mirror on its instance variable: #localSelectors. The localSelectors instance variable is a Set holding all selectors (method names) defined locally in that class. Since this is a collection of basic instances (i.e symbols), further processing on it (like printing) would be more convenient if instead of a mirror (i.e the ivMirror in this case) we had a local copy. This is achieved on line 3 where we send the message *resolveLocally* to the ivMirror. This message (at run-time) instructs the middleware to override its serialization policy specifically for this instance and return a local copy of the underlying remote reference.

Example Script 6: **Adapting serialization at run-time**

```
1 aClassMirror := aRemoteEnvironment globalAt: #Class.
2 ivMirror := aClassMirror objInstVarNamed: #localSelectors.
3 localSet := ivMirror resolveLocally.
```

7.2 Experimental Setting

For validating Mercury we have considered three different kinds of devices as debugging targets. These devices (see Figure 11) where chosen as illustrative examples of either:

- Targets that have different hardware or environment settings than development machines.
- Targets that are not locally or easily accessible.
- Targets that have no input/output interfaces for local development.

Through this setting, we have verified the applicability of Mercury for different debugging targets. We experimented on how a debugging session can benefit from Mercury's properties, by studying the following two use-cases:

1. Combining agile development [ABF05] with debugging **in a single remote debugging session** without the need of re-deployment.
2. Supporting both OO-centric [RBN12] and Stack-based debugging **in a remote setting** through remote object instrumentation.

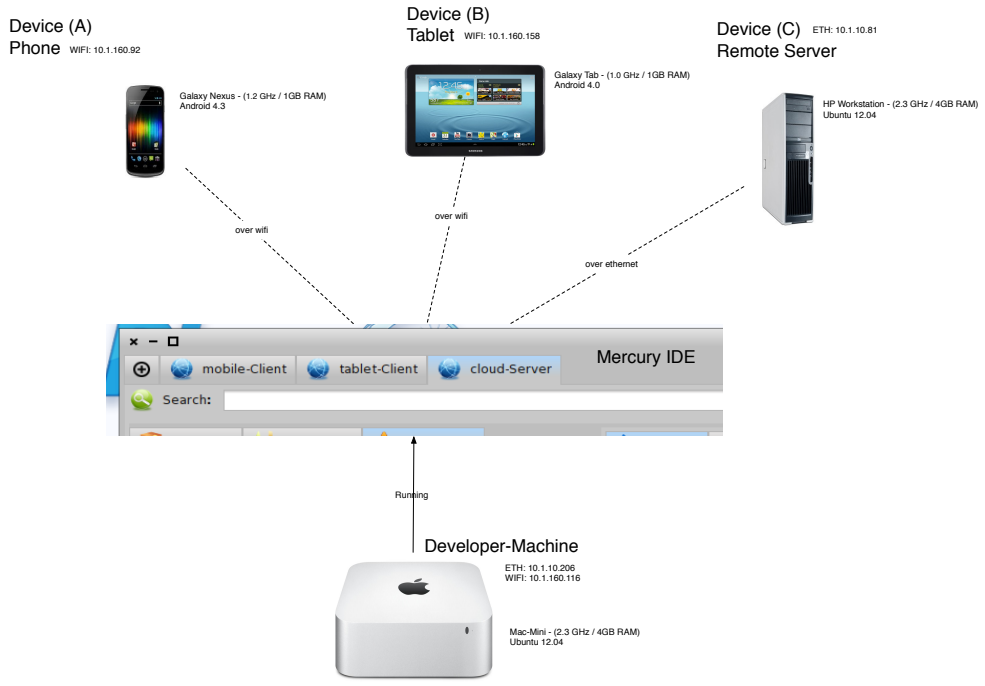


Figure 11 – Experimental Set-up for our Debugging Targets

Figure 11 shows the set-up of our experiment. In the upper part of the figure we depict our debugging targets. Device A is a smart-phone target connected to our development machine through wifi. Device B is a tablet target also connected through a wireless network, while Device C is a remote server to which we connect through ethernet.

In the lower part of the figure we show the development machine running our debugging front-end. A cropped screenshot of the Mercury IDE is shown at the center of the figure. Each tab corresponds to tools supporting remote development and debugging of a single target.

The developer machine connects to our targets through the two communication interfaces designated as *ETH and WIFI* for ethernet and wireless communication channels respectively. For our two android devices (phone and tablet), we have also tested communication through a usb channel that establishes ethernet connections using port forwarding. [5]

7.2.1 The Droid and Cloud File Browsers

Figure 17 shows the Droid-Browser application running on the phone and tablet devices (middle and right part of Figure 17 respectively). The Droid-Browser is a local file browsing application that presents the option to upload local files stored on the device to the cloud (i.e the remote server of our experimental setting).

On the other hand the Cloud-Browser is a normal web-application that presents the option to download files that where previously uploaded on the server.

The two applications share part of their code for file browsing and serving of web-pages as seen in Figure 12. The core logic of both applications resides in two subclasses of a common ancestor class named FileBrowser. FileBrowser is itself a subclass of WAComponent which is part of the Seaside [DLR04] web-framework.

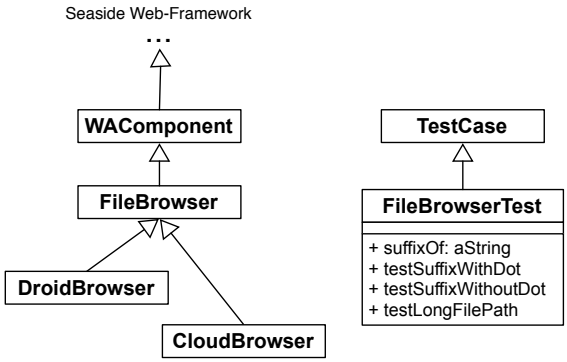


Figure 12 – **Left:** The Droid and Cloud browser apps implemented as Seaside components
Right: FileBrowserTest class and methods introduced through run-time evolution

7.3 Case Study I: Remote Agile Debugging

Our starting point is the deployment of our target software in all three devices, and the subsequent launch of the applications.

We then connect through the debugging front-end of Mercury to our remote targets. All our targets upon start-up raised the same error. From the exception name and the remote stack we can deduce that a NotFound error was triggered from inside the #detect: method of class Collection, as seen in Script 8.

Script 8: **The method which raised the initial error**

```
detect: aBlock
1       "Evaluate aBlock with each of the receiver's elements as the argument.
2       Answer the first element for which aBlock evaluates to true."
3
4       ^ self detect: aBlock ifNone: [self errorNotFound: aBlock]
```

After navigating the stack through our context mirrors we come to the first context related with our application, which can be seen in Script 9. The method in Script 9 is an extension

[5] http://developer.android.com/tools/help/adb.html

of our application for the system class String, which calculates the suffix of a given filename. Mercury informs us that the offending method call to #detect: originated from our code on line 3.

Script 9: **Calculating a filename suffix**

```
String>>suffix
1       | dot dotPosition |
2       dot := FileDirectory dot.
3       dotPosition := (self size to: 1 by:  -1) detect: [ :i | (self at: i) = dot ].
4       ^ self copyFrom: dotPosition + 1 to: self size
```

In turn the method suffix was called while the Droid Browser was trying to render a corresponding icon for a file system entry according to its suffix, as seen in Script 10. The method #renderPathOn: in Script 10 belongs to the class FileBrowser (superclass of both Droid and CloudBrowser as seen in Figure 12) which seems to be the reason why all of our targets failed to render their ui. To validate this hypothesis we check the stack on all 3 targets and browse the offending file system entries for each case:

Phone: '/charger'

Tablet: '/default.prop'

Server: '/var/www/User/.profile-xmind-portable-201212250029'

Script 10: **Icon rendering code calling the suffix method**

```
FileBrowser>>renderPathOn: html
[...]
        html image url: (FileIcons urlOf: (each asString suffix , 'Png') asSymbol).
[...]
```

The entries unfortunately tells us three different things: the suffix method fails both when it is invoked on a filename with no extension (as in the case of our smart-phone target) and on file paths that do have an extension (as in the case of our tablet target). Moreover in the case of the server target the failing filename is a longer file-path whose dot signifies something other than an extension (the fact that this is a hidden file on unix systems), which may be a contributing factor.

7.3.1 Remote Agile Debugging through Run-Time Evolution

Up until now we have seen a normal remote debugging session, where we where able to browse remote targets, navigate their stack and control execution. We will now see how we can use Mercury to dynamically introduce new code and tests while debugging without lengthy re-deployments of our applications.

By doing so we aim to achieve the following:

1. Re-produce the initial error multiple times in order to test different hypothesis without the need of re-deployment.
2. Simplify the offending context without re-starting the debugging session.
3. Maintain the state and suspended execution flow of the initial unhandled errors:
   (a) In order to cross-examine the initial failing state with new findings.
   (b) In case the initial errors are not easily reproducible (as is the case with heisenbugs [Gra86])

Our next step is shown in Figure 12. Since Mercury can dynamically evolve the target's code (run-time evolution) we can remotely introduce new classes and methods for testing to all of our targets while debugging.

In this case we introduce the test class FileBrowserTest (right part of Figure 12) as a subclass of the class TestCase which is part of the SUnit framework on the target. Getting a class mirror on FileBrowserTest will allows us to incrementally run tests on our remote machines and debug their results, without ever quitting our current debugging session.

The code of our FileBrowserTest class is given on Script 11. Our first method #suffixOf: is a helper method that replicates the behavior of the String»#suffix method of Script 9. Our second method #testSuffixWithDot invokes our helper method on a simple dotted filename and makes an assertion about the return value of this invocation (this case is similar to our initial error on our tablet). Method #testSuffixWithoutDot makes an assertion for the case of a not-dotted filename (similar to our initial error for the smart-phone). Finally testHiddenFilePath tests a long hidden filename with its full path (similar to our initial error on the cloud server). Note here that all three tests will raise a new exception both when our helper method has a defect as well as in the case of a failed assertion.

Script 11: **Test methods**

```
FileBrowserTest>>suffixOf: aString
     "assumes that I'm a file name, and answers my suffix, the part after the last dot"
     | dot dotPosition |
     dot := FileDirectory dot.
     dotPosition := (aString size to: 1 by:  -1) detect: [ :i | (aString at: i) = dot ].
     ^ aString copyFrom: dotPosition + 1 to: aString size

FileBrowserTest>>testSuffixWithDot
     self assert:
          (self suffixOf: 'filename.ext') = 'ext'

FileBrowserTest>>testSuffixWithoutDot
     self assert:
          (self suffixOf: 'filename') = ''

FileBrowserTest>>testHiddenFilePath
     self assert:
         (self suffixOf: '/var/www/User/.a-looooooooong-hidden-filename') =
         'a-looooooooong-hidden-filename'
```

We add our test class and methods to all three targets, and run the tests. This way we will be able to determine if there is some device-specific cause for the error on one of the devices (e.g., different representation of file-systems). This process is shown in Figure 13.

On Step 1 we run each test individually, on Step 2 we examine the results on the bottom left panel. If the results informs us of an error we can hit the debug button to examine and manipulate it (Step 3). Finally on Step 4 we can see that we are able to switch and cross-examine state and execution between the initial error and the re-produced errors from the test cases. Note here that it is possible to run and debug a single test multiple times, although in this case we run and debug each test only once.

We repeat this process for all three devices and find that all tests fail on all three of our targets. By doing so we deduce that there is no device-specific cause underlying each case, although each one of the tests may be failing for different reasons.

7.3.2 Debugging Hypotheses and Fixes

At this point having 12 different threads of execution at our disposal spanning 3 different devices, we can start debugging our hypotheses.

Our expectation for the String»suffix method is to return an empty string for filenames without a dot. Since our second test fails with a NotFound error notification we can now device

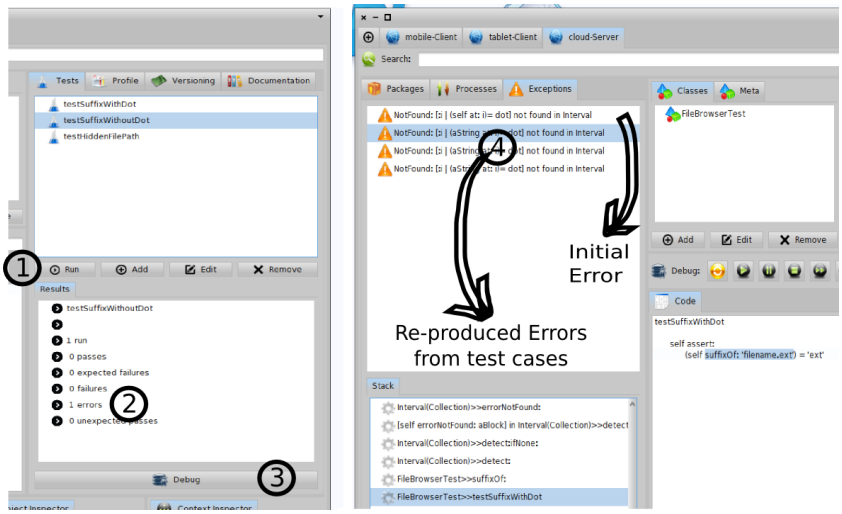


Figure 13 – Remotely Running and Debugging multiple Test-Cases while maintaining the initial error

a possible fix. We need to introduce an error handler for this case which will return the empty string when the error is raised. We want of course to test this possible fix before applying it to the String»suffix method, especially because we are not expecting that the fix will solve the two other cases of the defect.

In order to do so we remotely update the FileBrowserTest»suffixOf: method as seen in Script 12. In line 5 of Script 12 we introduce an #on:do: exception handler that returns an empty string when the NotFound error is raised. Subsequently we re-run all tests. The results are shown in Figure 14.

Script 12: **Updating the suffixOf: method**

```
FileBrowserTest>>suffixOf: aString
    [...]
4   dotPosition := [(aString size to: 1 by:  -1) detect: [ :i | (aString at: i) = dot ]]
5                  on: NotFound do: [ ^ ''].
6   ^ aString copyFrom: dotPosition + 1 to: aString size
```

In Figure 14 on the left we can see that our #testSuffixWithoutDot test now runs successfully, ensuring the applicability of our fix for this case. For the two other cases though as we expected the tests fail. After the introduction of the error handler in FileBrowserTest»suffixOf: the defect manifests itself as failed assertions on our two remaining tests. Debugging these last two failed assertions will be the focus of our second case study.

7.3.3 Results

Using our results shown in Figures 13 and 14 we can now verify that by being able to dynamically evolve the target's code (run-time evolution) we were able to introduce and debug tests **without lengthy application re-starts or re-deployments**.

7.4 Case Study II: Remote Object Instrumentation

7.4.1 Introduction

In Section 5 we saw how the Mercury model supports the instrumentation of semantical events in a remote setting. Our goal now in this second case study is two-fold:

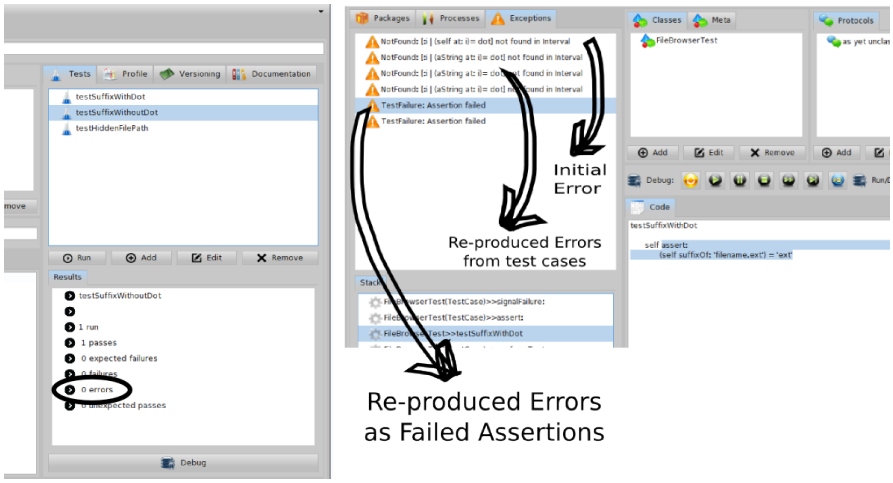


Figure 14 – Successfully debugged first failing test (left). Defect now manifests itself as failed assertions (right).

1. Verify that the remote object instrumentation facilities of Mercury can bring the idea of oo-centric debugging in a distributed setting.

2. Provide an example where Mercury uses the two paradigms (i.e stack-based and oo-centric debugging) in a complementary fashion.

7.4.2 The Hidden Path Hypothesis

We continue where we left off in our first case-study (Figure 14) but now turn our attention to the hidden path failure on the server that we discussed on Section 7.3.

We would now like to examine the execution of the FileBrowserTest»suffixOf: method more closely. We *restart* the execution of the current context and then we *step-into* the suffixOf: method (Script 11).

In order to get to our point of interest though we now have to follow the iteration seen on line 4 of Script 11 (inside the block closure argument to the #detect: method). Getting inside the loop requires in total 10 control-flow commands from our initial execution point and for each additional iteration 3 commands more. A breakpoint inside the loop can reduce the number of commands we need to issue from the ui (to 1 command per iteration), but still the placement of the dot is such that we would need 30 iterations to get there (the loop iterates the string starting from the end). So even setting a break-point will be time consuming, especially if we need to reproduced and re-examine the failed assertion several times. In addition we do not know the specific inner working of the #detect: method and if we continue using a stack-based approach we would need to step-into the #detect method between iterations.

7.4.3 Combining Object and Stack Debugging in a Remote Setting

Stack-based debugging got us this far, but it is becoming cumbersome. We need to narrow the domain of our examination. What is needed here is either a conditional break-point or a watch-point breaking exactly at the required iteration. A generalized object-oriented version of such facilities as we saw was proposed by Ressia [Res12]. In a nutshell these facilities instead of using source-code locations (line numbers or method-names) as their point of reference for interrupting execution, use run-time objects of interest and their events. For instance, the

programmer can put a breakpoint on the next message send that will be received by an object or the next read (or write) of one of its instance variable. We will now see how Mercury brings this object-centric debugging to a distributed setting through remote instrumentation.

We continue from our execution point inside the suffixOf: method (Script 11) but instead of trying to navigate through the loop or inside the #detect: method we now invoke the remote instrumentation interface of Mercury (seen in Figure 15).

In the left part of Figure 15 we can see the failed assertion and its stack, the execution is currently suspended in the FileBrowserTest»suffixOf: method. In the right we can see the remote instrumentation ui. From the panel in Step 1 we can choose the semantical event which we wish to instrument (Object Interaction in our case). The text entry in Step 2 receives an expression whose result will be returned as a mirror in the developers machine. This mirror will serve as a target for the semantical event of Step 1. The text entry in Step 3 receives additional information related to the event which we wish to instrument. In the case of the *Object Interaction* event we need to supply an additional expression for calculating the mirror of the interacting object. Finally on Step 4 we can supply an optional condition for the meta-action we wish to perform and the meta-action itself which will be triggered upon the semantical event defined by Steps 1 through 3.

The code for implementing the instrumentation's meta-action can be seen in Script 13. On line 1 we can see that the meta-action receives two arguments. The first argument (named reifications on our Script) represents meta-information relating to the event (such as the object that triggered a particular event), while the second argument reifies the reflectogram [TNCC03] of this particular meta-jump, which can control meta-level execution. On line 3 we instruct the execution on the remote target to halt in a context of the object that triggered the event. While on lines 4 through 6 we instruct the reflectogram to perform the default action (for this semantical event) when execution resumes from the breakpoint of line 3.

Script 13: **Object-centric conditional watchpoint meta-action**

```
1 [:reifications :reflectogram |
2
3       reifications trigger halt.
4       reflectogram
5               override: true;
6               returnValue: reflectogram defaultAction.
7
8 ]
```

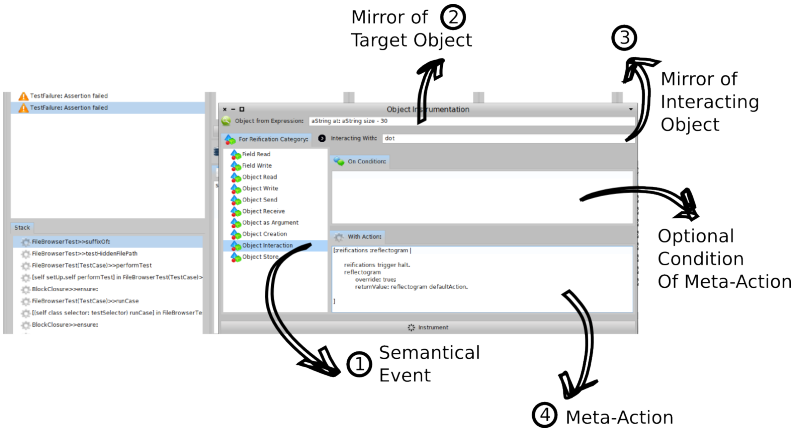


Figure 15 – Remote Object Instrumentation

In a nutshell we have instructed Mercury through this process to halt execution the next time the dot inside the hidden filepath will interact in anyway with the dot object (representing the

suffix separator defined by the FileSystem). This way through remote object instrumentation we have implemented a custom conditional watchpoint for our case with object-centric semantics. The results of this process are seen in Figure 16.

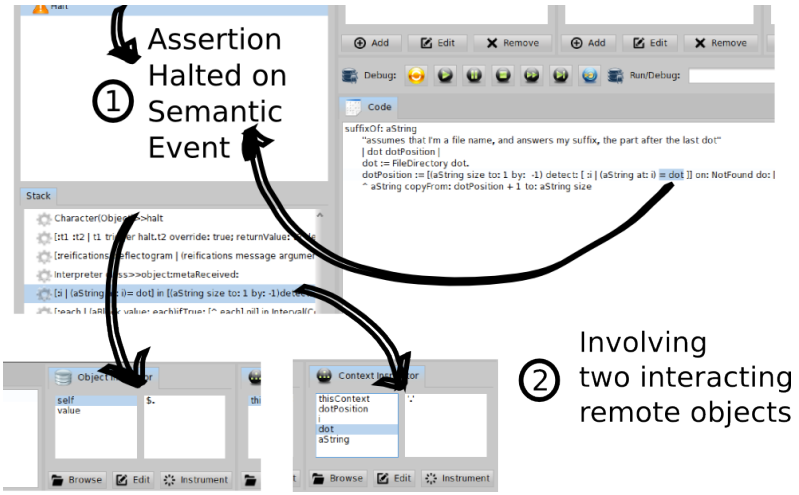


Figure 16 – Halting on Semantical Events

In the left part of Figure 16 we can see that the assertion's execution continued up until the semantical event that we defined above and then halted (Step 1). Specifically (as seen in the code editor of Figure 16) execution halted when the two objects we were targeting (the dot inside the filepath and the dot of the FileSystem interacted). The interaction took place while we were comparing the two objects (*(aString at: i) = dot*).

After examining the two objects we were targeting (Step 2) we pinpoint a mismatch. We are comparing two dots with different string representations (i.e $. and '.'). The defect causing our assertion to fail now becomes apparent: we are comparing a Character instance ($.) to a String instance ('.') which Pharo does not automatically cast.

We test our hypothesis by restarting execution on our suspended context and changing the code of the suffixOf: method compared to Script 12 as follows:

Script 14: **Fix applied to the suffixOf: method**

```
        [...]
3       dot := FileDirectory dot first.
        [...]
```

By resuming execution we now validate that the ui on all three targets is now rendering properly as seen in Figure 17. We show highlighted the filenames responsible for the initial defect.

7.4.4 Results

Using our results shown in Figures 15 and 16 we can now verify that through remote object instrumentation Mercury can support the idea of oo-centric debugging in a remote setting. More specifically we provided a real-world example where **in a single remote-debugging session** the two approaches (stack-based and oo-centric debugging) where used in complementary fashion to provide a custom mechanism for object-centric conditional watchpoints.

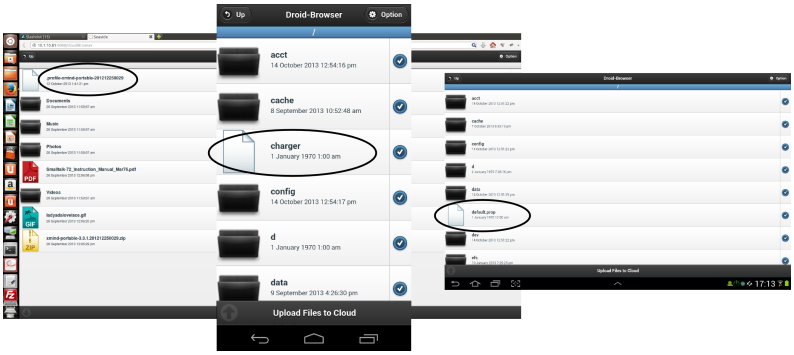


Figure 17 – Debugged applications on Server, Mobile and Tablet targets

8 Conclusion and Future Work

In this work we have proposed Mercury: a live mirror-based model and infrastructure for remote debugging. Mercury exhibits three desirable properties that we have identified as important for remote debugging, namely: *run-time evolution*, *semantic instrumentation*, and *adaptable distribution*. Run-time evolution is the ability of a remote debugging solution to incrementally update all parts of a remote application without losing the running context (i.e without stopping the application). Semantic instrumentation is the ability of a debugging solution to alter the semantics of a running process to assist debugging. Finally, adaptable distribution is the ability of a debugging solution to adapt its underlying middleware while debugging a remote target.

Mercury supports run-time evolution through a causal connection between the meta-level running on the developer machine, and the application to debug (the base-level) on the target device. The two levels are connected both computationally and structurally. It supports semantic instrumentation through the reification of the underlying execution environment (virtual-machine) inside the run-time environment of the target (as an interpreter). Finally adaptable distribution is supported through a modular architecture of the underlying middleware. We have validated the applicability of our proposal through a prototype implementation in the Pharo language. We have illustrated our approach through several working examples in an experimental setting of two case-studies.

Future Work A future perspective for this work is to examine the prerequisites of supporting advanced debugging facilities such as delta-debugging [Zel02] in a remote setting. We also plan to explore more issues of mirror-based systems in a remote setting such as ontological correspondence [BU04]. Finally we would like to extend our implementation to be completely independent from local reflection facilities on the target (such as the host compiler) as exemplified in our previous work with MetaTalk [PBD+11].

References

[ABF05] Alex Abacus, Mike Barker, and Paul Freedman. Using test-driven software development tools. *IEEE Software*, 22(2):88–91, 2005.

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, Boston, MA, USA, 1998.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. URL: http://pharobyexample.org/.

[Bei90a] Boris Beizer. *Software Testing Techniques*. Thomson Computer Press, 1990.

[Bei90b] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[Ben88] John K. Bennett. Distributed smalltalk: Inheritance and reactiveness in distributed systems, 1988.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the rescue. In *IN PROCEEDINGS ECOOP '98, VOLUME 1445 OF LNCS*, pages 396–417. Springer-Verlag, 1998.

[BH02] Padmanabhan Santhanam Brent Hailpern. Software debugging, testing, and verification. *IBM Systems Journal*, 2002.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of Adaptable and Extensible Component Systems*, Grenoble, France, November 2002.

[BNVC+11] Elisa Gonzalez Boix, Carlos Noguera, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt. Reme-d: A reflective epidemic message-oriented debugger for ambient-oriented applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1275–1281, New York, NY, USA, 2011. ACM. doi:10.1145/1982185.1982463.

[Bru12] Eric Bruno. A long look at jvm languages. http://www.drdobbs.com/jvm/a-long-look-at-jvm-languages/240007765, 2012.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press. URL: http://bracha.org/mirrors.pdf.

[Cat14] John Catherino. The cajo project. https://java.net/projects/cajo/pages/Home, 2014.

[CN03] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *In Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03)*, volume 2830 of *LNCS*, pages 364–376, 2003.

[Cut14] Tom Van Cutsem. Ambient-oriented programming – reflective programming. http://soft.vub.ac.be/amop/at/tutorial/reflection, 2014.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007. URL: http://rmod.lille.inria.fr/archives/papers/Denk07b-TOOLS07-Submethod.pdf.

[DL02] Pierre-Charles David and Thomas Ledoux. An infrastructure for adaptable middleware. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 773–790. Springer Berlin Heidelberg, 2002. URL: http://dx.doi.org/10.1007/3-540-36124-3_52.

[DLR04] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of 12th International Smalltalk Conference (ISC'04)*, pages 231–257, September 2004. URL: http://scg.unibe.ch/archive/papers/Duca04eSeaside.pdfhttp://www.iam.unibe.ch/publikationen/techreports/2004/iam-04-008.

[Dra14] Iulian Dragos. Scala ide documentation – scala debugger. http://scala-ide.org/docs/current-user-doc/features/scaladebugger/index.html, 2014.

[Eis14] Andrew Eisenberg. New groovy debug support in sts 2.5.1. http://spring.io/blog/2010/11/30/new-groovy-debug-support-in-sts-2-5-1/, 2014.

[Fer89] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.

[Gol84] Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[Goo14] Google. Debugging javascript. https://developer.chrome.com/devtools/docs/javascript-debugging, 2014.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[Gra14] David Grayson. Using the jruby debugger. https://github.com/jruby/jruby/wiki/UsingTheJRubyDebugger, 2014.

[Hum99] Watts S. Humphrey. Bugs or defects ? *Technical Report Vol. 2, Issue 1*, 1999.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997. URL: http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html, doi:10.1145/263700.263754.

[Joy14] Inc Joyent. Node.js v0.10.32 manual & documentation. http://nodejs.org/api/debugger.html, 2014.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, June 2002. URL: http://doi.acm.org/10.1145/508448.508470, doi:10.1145/508448.508470.

[Led99] Thomas Ledoux. Opencorba: a reflective open broker. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 197–214. Springer Berlin Heidelberg, 1999. URL: http://dx.doi.org/10.1007/3-540-48443-4_19, doi:10.1007/3-540-48443-4_19.

[LP90] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk: vol. 1*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[Mar06] Philippe Marschall. Persephone: Taking Smalltalk reflection to the submethod level. Master's thesis, University of Bern, December 2006. URL: http://scg.unibe.ch/archive/masters/Mars06a.pdf.

[McA95] Jeff McAffer. Meta-level programming with coda. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.

[MCTT07] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Eric Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings the ACM Dynamic Languages Symposium (DLS 2007)*, October 2007.

[Mic12a] James Mickens. Rivet: Browser-agnostic remote debugging for web applications. In *USENIX Annual Technical Conference*, pages 333–345, 2012.

[Mic12b] Microsoft. How to: Set up remote debugging, visual studio 2012. http://msdn.microsoft.com/en-us/library/bt727f1t.aspx, 2012.

[Mic12c] Microsoft. Supported code changes (c#), visual studio 2012. http://msdn.microsoft.com/en-us/library/ms164927.aspx, 2012.

[Mic13] Microsoft. Setting up remote debugging, visual studio 2013. http://msdn.microsoft.com/en-us/library/bt727f1t%28v=vs.71%29.aspx, 2013.

[Mic14] Microsoft. Using the f12 developer tools to debug javascript errors. http://msdn.microsoft.com/en-us/library/ie/gg699336(v=vs.85).aspx, 2014.

[Mir08] Eliot Miranda. Cog blog. speeding up croquet and squeak with a new open-source vm from qwaq, 2008. URL: http://www.mirandabanda.org/cogblog/.

[MO06] Sean McDirmid and Martin Odersky. The scala plugin for eclipse. In *Proceedings of Workshop on Eclipse Technology eXchange (ETX)*, 2006.

[Moz14] Mozilla. Mdn – debugging javascript. https://developer.mozilla.org/en/docs/Debugging_JavaScript, 2014.

[MVCT+09] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in ambienttalk. *Softw. Pract. Exper.*, 39(7):661–699, May 2009. doi:10.1002/spe.v39:7.

[OL14] Oracle Oracle Labs. The maxine inspector. https://wikis.oracle.com/display/MaxineVM/The+Maxine+Inspector, 2014.

[Ora13a] Oracle. Java debug interface (jdi). http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html, 2013.

[Ora13b] Oracle. Java platform debugger architecture (jpda). http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/, 2013.

[Pap13] Nikolaos Papoulias. *Remote Debugging and Reflection in Resource Constrained Devices*. These, Université des Sciences et Technologie de Lille - Lille I, December 2013.

[PBD+11] Nikolaos Papoulias, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Towards structural decomposition of reflection with mirrors. In *Proceedings of International Workshop on Smalltalk Technologies (IWST'11)*, Edingburgh, United Kingdom, 2011. URL: http://hal.inria.fr/inria-00629175/en/.

[PTP07] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, 42(10):535–552, 2007. doi:10.1145/1297105.1297067.

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012. URL: http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf, doi:10.1109/ICSE.2012.6227167.

[RC00] Barry Redmond and Vinny Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for java. In *Proceedings of European Conference on Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures*, 2000.

[Res12] Jorge Ressia. *Object-Centric Reflection*. PhD thesis, Institut fur Informatik und angewandte Mathematik, 2012.

[Riv96] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.

[RKC01] Manuel Rom, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001. doi:http://doi.ieeecomputersociety.org/10.1109/MDSO.2001.5.

[RN01] Michael Richmond and James Noble. Reflections on remote reflection. In *Proceedings of the 24th Australasian Conference on Computer Science*, ACSC '01, pages 163–170, Washington, DC, USA, 2001. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=545564.545585.

[RRGN10] Jorge Ressia, Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Run-time evolution through explicit meta-objects. In *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 37–48, October 2010. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/. URL: http://scg.unibe.ch/archive/papers/Ress10a-RuntimeEvolution.pdf.

[RS03] Stan Shebs Richard Stallman, Roland Pesch. *Debugging with GDB*. Gnu Press, 2003.

[SMDV06] J. Sillito, G.C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th International Symposium on Foundations on Software Engineering*, SIGSOFT '06/FSE-14, pages 23–34. ACM, 2006.

[Som01] Ian Sommerville. *Software Engineering (6th ed.)*. Addison-Wesley, 2001.

[TBN01] Éric Tanter, Noury Bouraqadi, and Jacques Noyé. Reflex — towards an open reflective extension of Java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 25–43. Springer-Verlag, 2001.

[TCD13] Camille Teruel, Damien Cassou, and Stéphane Ducasse. Object Graph Isolation with Proxies. In *DYLA - 7th Workshop on Dynamic Languages and Applications, Collocated with 26th European Conference on Object-Oriented Programming - 2013*, Montpellier, France, 2013.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003. URL: http://www.dcc.uchile.cl/~etanter/research/publi/2003/tanter-oopsla03.pdf.

[USA05] David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM. doi:10.1145/1094855.1094865.

[WAC+98] Lauren Wood, Vidur Apparao, Laurence Cable, Mike Champion, Mark Davis, Joe Kesselman, Tom Pixley, Jonathan Robie, Peter Sharpe, and Chris Wilson. Document object model (dom) level 1 specification. *w3C recommendation*, 1, 1998.

[WHV+12] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynes, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. Technical Report 2012-0098, Oracle Labs, 2012.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10. ACM, 2010.

[Zel02] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM Press. doi:10.1145/587051.587053.

[Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.

[Zer11] ZeroTurnAround. Java ee productivity report 2011. http://zeroturnaround.com/wp-content/uploads/2010/11/Java_EE_Productivity_Report_2011_finalv2.pdf, 2011.

[Zer12] ZeroTurnAround. What developers want: The end of application re-deploys. http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf, 2012.

Appendices

A Detailed Evaluation of Existing Solutions and Comparison

Property		JPDA/JVM	.NET	GNU-DEBUGGER	DCE	JREBEL	SMALLTALK
Run-Time Evolution	Add/Rem Packages	✗	✗	✗	✓	✓	✓
	Add/Rem Classes/Prototype	✗	✗	✗	✓	✓	✓
	Add/Rem IVs	✗	✗	✗	✓	✓	✓
	Add/Rem Methods	✗	✗	✗	✓	✓	✓
	Method (Body) HotSwapping	✓	✓	✗	✓	✓	✓
	Hierarchy/Delegation Editing	✗	✗	✗	✓	✓	✓
Sem. Instrumentation	Method Execution	✓	✓	✓	✓	✓	✓
	Statement Execution	✓	✓	✓	✓	✓	✓
	Field Read	✓	✗	✗	✓	✓	✗
	Field Write	✓	✗	✗	✓	✓	✗
	Object Read	✗	✗	✗	✗	✗	✗
	Object Write	✗	✗	✗	✗	✗	✗
	Object Send	✗	✗	✗	✗	✗	✗
	Object Receive	✗	✗	✗	✗	✗	✗
	Object as Argument	✗	✗	✗	✗	✗	✗
	Object Creation	✗	✗	✗	✗	✗	✗
	Object Interaction	✗	✗	✗	✗	✗	✗
	Object Stored	✗	✗	✗	✗	✗	✗
	Condition/Action	✗	✗	✓	✗	✗	✓
Ad. Distribution		+	++	+	+	-	-

Property		JS/RIVET	TOD	AMBIENTTALK	MAXINE	BIFROST	MERCURY
Run-Time Evolution	Add/Rem Packages	✓	✗	✗	✗	✓	✓
	Add/Rem Classes/Prototypes	✓	✗	✗	✗	✓	✓
	Add/Rem IVs	✓	✗	✓	✗	✓	✓
	Add/Rem Methods	✓	✗	✗	✗	✓	✓
	Method (Body) HotSwapping	✓	✓	✓	✓	✓	✓
	Hierarchy/Delegation Editing	✓	✗	✗	✗	✓	✓
Sem. Instrumentation	Method Execution	✓	✓	✓	✓	✓	✓
	Statement Execution	✓	✓	✓	✓	✓	✓
	Field Read	✗	✓	✓	✓	✓	✓
	Field Write	✗	✓	✓	✓	✓	✓
	Object Read	✗	✓	✓	✓	✓	✓
	Object Write	✗	✓	✓	✓	✓	✓
	Object Send	✗	✗	✗	✗	✓	✓
	Object Receive	✗	✓	✓	✓	✓	✓
	Object as Argument	✗	✗	✗	✗	✓	✓
	Object Creation	✗	✗	✓	✗	✓	✓
	Object Interaction	✗	✗	✗	✗	✓	✓
	Object Stored	✗	✗	✗	✓	✗	✓
	Condition/Action	✗	✗	✗	✗	✓	✓
Ad. Distribution		++	+	+++	+	-	+++

Table 3 – Properties evaluation of Mercury and existing debugging solutions

[6] Java-based solutions have condition-only support